Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 174 (2007) 23–37
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

# An Abstract Decision Procedure for Satisfiability in the Theory of Recursive Data Types

Clark Barrett[a] Igor Shikanian[a] Cesare Tinelli[b]

[a] *New York University,* {barrett,igor}@cs.nyu.edu

[b] *University of Iowa,* tinelli@cs.uiowa.edu

**Abstract**

The theory of recursive data types is a valuable modeling tool for software verification. In the past, decision procedures have been proposed for both the full theory and its universal fragment. However, previous work has been limited in various ways. In this paper, we present a general algorithm for the universal fragment. The algorithm is presented declaratively as a set of abstract rules which are terminating, sound, and complete. We show how other algorithms can be realized as strategies within our general framework. Finally, we propose a new strategy and give experimental results showing that it performs well in practice.

*Keywords:* recursive data types, decision procedures, term algebras, satisfiability modulo theories

## 1 Introduction

Recursive data types are commonly used in programming. The same notion is also a convenient abstraction for common data types such as records and data structures such as linked lists used in more conventional programming languages. The ability to reason automatically and efficiently about recursive data types thus provides an important tool for the analysis and verification of programs.

Perhaps the best-known example of a simple recursive data type is the *list* type used in LISP. Lists are either the *null* list or are constructed from other lists using the *constructor cons*. This constructor takes two arguments and returns the result of prepending its first argument to the list in its second argument. In order to retrieve the elements of a list, a pair of *selectors* is provided: *car* returns the first element of a list and *cdr* returns the rest of the list.

More generally, we are interested in any set of (possibly mutually) recursive data types, each of which contains one or more constructors. Each constructor has selectors that can be used to retrieve the original arguments as well as a *tester*

1571-0661 
doi:10.1016/j.entcs.2006.11.037

which indicates whether a given term was constructed using that constructor. As an example of the more general case, suppose we want to model lists of trees of natural numbers. Consider a set of three recursive data types: *nat*, *list*, and *tree*. *nat* has two constructors: *zero*, which takes no arguments (we call such a constructor a *nullary* constructor or *constant*); and *succ*, which takes a single argument of type *nat* and has the corresponding selector *pred*. The *list* type is as before except that we now specify that the elements of the list are of type *tree*. The *tree* type in turn has two constructors: *node*, which takes an argument of type *list* and has the corresponding selector *children*, and *leaf*, which takes an argument of type *nat* and has the corresponding selector *data*. We can represent this set of types using the following convenient notation based on that used in functional programming languages:

$$nat := succ(pred : nat) \mid zero;$$

$$list := cons(car : tree,\ cdr : list) \mid null;$$

$$tree := node(children : list) \mid leaf(data : nat);$$

The testers for this set of data types are *is_succ*, *is_zero*, *is_cons*, *is_null*, *is_node*, and *is_leaf*.

Propositions about a set of recursive data types can be captured in a sorted first-order language which closely resembles the structure of the data types themselves in that it has function symbols for each constructor and selector, and a predicate symbol for each tester. For instance, propositions that we would expect to be true for the example above include: (i) $\forall x : nat.\ succ(x) \neq zero$; (ii) $\forall x : list.\ x = null \vee is\_cons(x)$; and (iii) $\forall x : tree.\ is\_leaf(x) \rightarrow (data(x) = zero \vee is\_succ(data(x)))$.

In this paper, we discuss a procedure for deciding such formulas. We focus on satisfiability of a set of literals, which (through well-known reductions) can be used to decide the validity of universal formulas.

There are three main contributions of this work over earlier work on the topic. First, our setting is more general: we allow mutually recursive types and multiple constructors. The second contribution is in presentation. We present the theory itself in terms of an initial model rather than axiomatically as is often done. Also, the presentation of the decision procedure is given as abstract rewrite rules, making it more flexible and easier to analyze than if it were given imperatively. Finally, as described in Section 4, the flexibility provided by the abstract algorithm allows us to describe a new strategy with significantly improved practical efficiency.

**Related Work.**

Term algebras over constructors provide the natural intended model for recursive data types. In [7] two dual axiomatizations of term algebras are presented, one with constructors only, the other with selectors and testers only.

An often-cited reference for the quantifier-free case is the treatment by Nelson and Oppen in 1980[11,12] (where the problem is also shown to be NP-complete). In particular, Oppen's algorithm in [12] gives a detailed decision procedure for a

single recursive data type with a single constructor; however, the case of multiple constructors is discussed only briefly and not rigorously.

More recently, several papers by Zhang et al. [14,15] explore decision procedures for a single recursive data type. These papers focus on ambitious schemes for quantifier elimination and combinations with other theories. A possible extension of Oppen's algorithm to the case of multiple constructors is discussed briefly in [14]. A comparison of our algorithm with that of [14] is made in Section 4.

Finally, another approach based on first-order reasoning with the superposition calculus is described in [5]. This work shows how a decision procedure for a recursive data type can be automatically inferred from the first-order axioms, even though the axiomatization is infinite. Although the results are impressive from a theoretical point of view, the scope is limited to theories with a single constructor and the practical efficiency of such a scheme has yet to be shown.

## 2 The Theory of Recursive Data Types

Previous work on recursive data types (RDTs) uses first-order axiomatizations in an attempt to capture the main properties of a recursive data type and reason about it. We find it simpler and cleaner to use a semantic approach instead, as is done in algebraic specification. A set of RDTs can be given a simple equational specification over a suitable signature. The intended model for our theory can be formally, and uniquely, defined as the initial model of this specification. Reasoning about a set of RDTs then amounts to reasoning about formulas that are true in this particular initial model.

### *2.1 Specifying RDTs*

We formalize RDTs in the context of many-sorted equational logic (see [9] among others). We will assume that the reader is familiar with the basic notions in this logic, and also with basic notions of term rewriting.

We start with the theory signature. We assume a many-sorted signature $\Sigma$ whose set of sorts consists of a distinguished sort $\mathsf{bool}$ for the Booleans, and $p \geq 1$ sorts $\tau_1, \ldots, \tau_p$ for the RDTs. We also allow $r \geq 0$ additional (non-RDT) sorts $\sigma_1, \ldots, \sigma_r$. We will denote by $s$, possibly with subscripts and superscripts, any sort in the signature other than $\mathsf{bool}$, and by $\sigma$ any sort in $\{\sigma_1, \ldots, \sigma_r\}$.

As mentioned earlier, the function symbols in our theory signature correspond to the constructors, selectors, and testers of the set of RDTs under consideration. We assume for each $\tau_i$ $(1 \leq i \leq p)$ a set of $m_i \geq 1$ *constructors* of $\tau_i$. We denote these symbols as $C^i_j$, where $j$ ranges from 1 to $m_i$. We denote the arity of $C^i_j$ as $n^i_j$ (0-arity constructors are also called nullary constructors or constants) and its sort as $s^i_{j,1} \times \cdots \times s^i_{j,n^i_j} \to \tau_i$. For each constructor $C^i_j$, we have a set of *selectors*, which we denote as $S^i_{j,k}$, where $k$ ranges from 1 to $n^i_j$, of sort $\tau_i \to s^i_{j,k}$. Finally, for each constructor, there is a *tester* [1] $isC^i_j : \tau_i \to \mathsf{bool}$.

[1] To simplify some of the proofs, and without loss of generality, we use functions to $\mathsf{bool}$ instead of

In addition to these symbols, we also assume that the signature contains two constants, true and false of sort bool, and an infinite number of distinct constants of each sort $\sigma$. The constants are meant to be names for the elements of that sort, so for instance if $\sigma_1$ were a sort for the natural numbers, we could use all the numerals as the constants of sort $\sigma_1$. Having all these constants in the signature is not necessary for our approach, but in the following exposition it provides an easy way of ensuring that the sorts in $\sigma$ are infinite. Section 5.1 shows that our approach can be easily extended to the case in which some of these sorts are finite. To summarize, the set of function symbols of the signature $\Sigma$ consists of:

$C^i_j : s^i_{j,1} \times \cdots \times s^i_{j,n^i_j} \to \tau_i$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i$,
$S^i_{j,k} : \tau_i \to s^i_{j,k}$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i,\ k = 1, \ldots, n^i_j$,
$isC^i_j : \tau_i \to \mathsf{bool}$, for $i = 1, \ldots, p,\ j = 1, \ldots, m_i$,
$\mathsf{true} : \mathsf{bool},\ \mathsf{false} : \mathsf{bool}$,
An infinite number of constants for each $\sigma_l$, for $l = 1, \ldots, r$.

As usual in many-sorted equational logic, we also have $p + r + 1$ equality symbols (one for each sort mentioned above), all written as $\approx$.

Our procedure requires one additional constraint on the set of RDTs: It must be *well-founded*. Informally, this means that each sort must contain terms that are not cyclic or infinite. More formally, we have the following definitions by simultaneous induction over constructors and sorts: (i) a constructor $C^i_j$ is well-founded if all of its argument sorts are well-founded; (ii) the sorts $\sigma_1, \ldots, \sigma_r$ are all well-founded; (iii) a sort $\tau_i$ is well-founded if at least one of its constructors is well-founded. We require that every sort be well-founded according to the above definition.

In some cases, it will be necessary to distinguish between *finite* and *infinite* $\tau$-sorts: (i) a constructor is *finite* if it is nullary or if all of its argument sorts are finite; (ii) a sort $\tau_i$ is *finite* if all of its constructors are finite, and is *infinite* otherwise; (iii) the sorts $\sigma_1, \ldots, \sigma_r$ are all infinite. As we will see, consistent with the above terminology, our semantics will interpret finite, resp. infinite, $\tau$-sorts indeed as finite, resp. infinite, sets.

We denote by $\mathcal{T}(\Sigma)$ the set of well-sorted ground terms of signature $\Sigma$ or, equivalently, the (many-sorted) term algebra over that signature. The RDTs with functions and predicates denoted by the symbols of $\Sigma$ are specified by the following set $\mathcal{E}$ of (universally quantified) equations. For reasons explained below, we assume that associated with every selector $S^i_{j,k} : \tau_i \to s^i_{j,k}$ is a distinguished ground term of sort $s^i_{j,k}$ containing no selectors (or testers), which we denote by $t^i_{j,k}$.

predicates for the testers.

**Equational Specification of the RDT:** for $i = 1, \ldots, p$:

$$\forall x_1, \ldots, x_{n_j^i}.\ isC_j^i(C_j^i(x_1, \ldots, x_{n_j^i})) \approx \mathsf{true} \quad (\text{for } j = 1, \ldots, m_i)$$

$$\forall x_1, \ldots, x_{n_{j'}^i}.\ isC_j^i(C_{j'}^i(x_1, \ldots, x_{n_{j'}^i})) \approx \mathsf{false} \quad (\text{for } j, j' = 1, \ldots, m_i,\ j \neq j')$$

$$\forall x_1, \ldots, x_{n_j^i}.\ S_{j,k}^i(C_j^i(x_1, \ldots, x_{n_j^i})) \approx x_k \quad (\text{for } k = 1, \ldots, n_j^i,\ j = 1, \ldots, m_i)$$

$$\forall x_1, \ldots, x_{n_{j'}^i}.\ S_{j,k}^i(C_{j'}^i(x_1, \ldots, x_{n_{j'}^i})) \approx t_{j,k}^i \quad (\text{for } j, j' = 1, \ldots, m_i,\ j \neq j')$$

The last axiom specifies what happens when a selector is applied to the "wrong" constructor. Note that there is no obviously correct thing to do in this case since it would correspond to an error condition in a real application. Our axiom specifies that in this case, the result is the designated ground term for that selector. This is different from other treatments (such as [7,14,15]) where the application of a wrong selector is treated as the identity function. The main reason for this difference is that identity function would not always be well-sorted in multi-sorted logic.

By standard results in universal algebra we know that $\mathcal{E}$ admits an *initial model* $\mathcal{R}$ and we can show the following result: [2] Let $\Omega$ be the signature obtained from $\Sigma$ by removing the selectors and the testers; then, the reduct of $\mathcal{R}$ to $\Omega$ is isomorphic to $\mathcal{T}(\Omega)$. Informally, this means that $\mathcal{R}$ does in fact capture the set of RDTs in question, as we can take the carrier of $\mathcal{R}$ to be the term algebra $\mathcal{T}(\Omega)$.

## 3 The Decision Procedure

In this section, we present a decision procedure for the satisfiability of sets of literals over $\mathcal{R}$. Our procedure builds on the algorithm by Oppen [12] for a single type with a single constructor. As an example of Oppen's algorithm, consider the *list* data type without the *null* constructor and the following set of literals: $\{cons(x, y) \approx z, car(w) \approx x, cdr(w) \approx y, w \not\approx z\}$. Oppen's algorithm uses a graph which relates terms according to their meaning in the intended model. Thus, $cons(x, y)$ is a parent of $x$ and $y$ and $car(w)$ and $cdr(w)$ are children of $w$. The equations induce an equivalence relation on the nodes of the graph. The Oppen algorithm proceeds by performing *upwards* (congruence) and *downwards* (unification) closure on the graph and then checking for cycles [3] or for a violation of any disequalities. For our example, upwards closure results in the conclusion $w \approx z$, which contradicts the disequality $w \not\approx z$.

As another example, consider the set of literals: $\{cons(x, y) \approx z, car(w) \approx x, cdr(w) \approx y, v \approx w, y \not\approx cdr(v)\}$. The new graph has a node for $v$, with $cdr(v)$ as its right child. The Oppen algorithm requires that every node with at least one child have a complete set of children, so $car(v)$ is added as a left child of $v$. Now, down-

[2] Proofs of all results in this paper can be found in [4].

[3] A simple example of a cycle is: $cdr(x) \approx x$.

wards closure forces $car(v) \approx car(w) \approx x$ and $cdr(v) \approx cdr(w) \approx y$, contradicting the disequality $y \not\approx cdr(v)$.

An alternative algorithm for the case of a single constructor is to introduce new terms and variables to replace variables that are inside of selectors. For example, for the first set of literals above, we would introduce $w \approx cons(s,t)$ where $s,t$ are new variables. Now, by substituting and collapsing applications of selectors to constructors, we get $\{cons(x,y) \approx z, w \approx cons(s,t), x \approx s, t \approx y, w \not\approx z\}$. In general, this approach only requires downwards closure.

Unfortunately, with the addition of more than one constructor, things are not quite as simple. In particular, the simple approach of replacing variables with constructor terms does not work because one cannot establish *a priori* whether the value denoted by a given variable is built with one constructor or another. A simple extension of Oppen's algorithm for the case of multiple constructors is proposed in [14]. The idea is to first guess a *type completion*, that is, a labeling of every variable by a constructor, which is meant to constrain a variable to take only values built with the associated constructor. Once all variables are labeled by a single constructor, the Oppen algorithm can be used to determine if the constraints can be satisfied under that labeling. Unfortunately, the type completion guess can be very expensive in practice.

Our presentation combines ideas from all of these algorithms as well as introducing some new ones. There is a set of upward and downward closure rules to mimic Oppen's algorithm. The idea of a type completion is replaced by a set of labeling rules that can be used to refine the set of possible constructors for each term (in particular, this allows us to delay guessing as long as possible). And the notion of introducing constructors and eliminating selectors is captured by a set of selector rules. In addition to the presentation, one of our key contributions is to provide precise side-conditions for when case splitting is necessary as opposed to when it can be delayed. The results given in Section 4 show that with the right strategy, significant gains in efficiency can be obtained.

We describe our procedure formally in the following, as a set of derivation rules. We build on and adopt the style of similar rules for abstract congruence closure [1] and syntactic unification [8].

### *3.1 Definitions and Notation*

In the following, we will consider well-sorted formulas over the signature $\Sigma$ above and an infinite set $X$ of variables. To distinguish these variables, which can occur in formulas given to the decision procedure described below, from other internal variables used by the decision procedure, we will sometimes call the elements of $X$ *input* variables.

Given a set $\Gamma$ of literals over $\Sigma$ and variables from $X$, we wish to determine the satisfiability of $\Gamma$ in the algebra $\mathcal{R}$. We will assume for simplicity, and with no loss of generality, that the only occurrences of terms of sort $\mathsf{bool}$ are in atoms of the form $isC_k^j(t) \approx \mathsf{true}$, which we will write just as $isC_k^j(t)$.

Following [1], we will make use of the sets $V_{\tau_i}$ ($V_{\sigma_i}$) of *abstraction* variables of

sort $\tau_i$ ($\sigma_i$); abstraction variables are disjoint from input variables (variables in $\Gamma$) and function as equivalence class representatives for the terms in $\Gamma$. We assume an arbitrary, but fixed, well-founded ordering $\succ$ on the abstraction variables that is total on variables of the same sort. We denote the set of all variables (both input and abstraction) in $E$ as $\mathcal{V}ar(E)$. We will use the expression $lbls(\tau_i)$ for the set $\{C_1^i, \ldots, C_{m_i}^i\}$ and define $lbls(\sigma_l)$ to be the empty set of labels for each $\sigma_l$. We will write $sort(t)$ to denote the sort of the term $t$.

The rules make use of three additional constructs that are not in the language of $\Sigma$: $\rightarrow$, $\mapsto$, and *Inst*.

The symbol $\rightarrow$ is used to represent *oriented* equations. Its left-hand side is a $\Sigma$-term $t$ and its right-hand side is an abstraction variable $v$. The symbol $\mapsto$ denotes *labellings* of abstraction variables with sets of constructor symbols. It is used to keep track of possible constructors for instantiating a $\tau_i$ variable. [4] Finally, the *Inst* construct is used to track applications of the **Instantiate** rules given below. It is needed to ensure termination by preventing multiple applications of the same **Instantiate** rule. It is a unary predicate that is applied only to abstraction variables.

Let $\Sigma^{\mathrm{C}}$ denote the set of all constant symbols in $\Sigma$, including nullary constructors. We will denote by $\Lambda$ the set of all possible literals over $\Sigma$ and input variables $X$. Note that this does not include oriented equations ($t \rightarrow v$), labeling pairs ($v \mapsto L$), or applications of *Inst*. In contrast, we will denote by $E$ multisets of literals of $\Lambda$, oriented equations, labeling pairs, and applications of *Inst*. To simplify the presentation, we will consistently use the following meta-variables: $c, d$ denote constants (elements of $\Sigma^{\mathrm{C}}$) or input variables from $X$; $u, v, w$ denote abstraction variables; $t$ denotes a *flat term*—i.e., a term all of whose proper sub-terms are abstraction variables—or a label set, depending on the context. $\boldsymbol{u}, \boldsymbol{v}$ denote possibly empty sequences of abstraction variables; and $\boldsymbol{u} \rightarrow \boldsymbol{v}$ is shorthand for the set of oriented equations resulting from pairing corresponding elements from $\boldsymbol{u}$ and $\boldsymbol{v}$ and orienting them so that the left hand variable is greater than the right hand variable according to $\succ$. Finally, $v \bowtie t$ denotes any of $v \approx t$, $t \approx v$, $v \not\approx t$, $t \not\approx v$, or $v \mapsto t$. To streamline the notation, we will sometimes denote function application simply by juxtaposition.

Each rule consists of a premise and one or more conclusions. Each premise is made up of a multiset of literals, oriented equations, labeling pairs, and applications of *Inst*. Conclusions are either similar multisets or $\bot$, where $\bot$ represents a trivially unsatisfiable formula. The soundness of our rule-based procedure depends on the fact that the premise $E$ of a rule is satisfied in $\mathcal{R}$ by a valuation $\alpha$ of $\mathcal{V}ar(E)$ iff one of the conclusions $E'$ of the rule is satisfied in $\mathcal{R}$ by an extension of $\alpha$ to $\mathcal{V}ar(E')$.

### *3.2 The derivation rules*

Our decision procedure consists of the following derivation rules on multisets $E$.

[4] To simplify the writing of the rules, some rules may introduce labeling pairs for variables with a non-$\tau$ sort, even though these play no role.

*Abstraction rules*

**Abstract 1** $$\frac{p[c],\ E}{c \to v,\ v \mapsto \mathit{lbls}(s),\ p[v],\ E} \quad \text{if} \quad \begin{array}{l} p \in \Lambda,\ c : s, \\ v \text{ fresh from } V_s \end{array}$$

**Abstract 2** $$\frac{p[C^i_j \boldsymbol{u}],\ E}{C^i_j \boldsymbol{u} \to v,\ p[v],\ v \mapsto \{C^i_j\},\ E} \quad \text{if } \ p \in \Lambda,\ v \text{ fresh from } V_{\tau_i}$$

**Abstract 3** $$\frac{p[S^i_{j,k} u],\ E}{\begin{array}{l} S^i_{j,1} u \to v_1, \ldots, S^i_{j,n^i_j} u \to v_{n^i_j},\ p[v_k], \\ v_1 \mapsto \mathit{lbls}(s_1), \ldots, v_{n^i_j} \mapsto \mathit{lbls}(s_{n^i_j}), E \end{array}} \quad \text{if} \quad \begin{array}{l} p \in \Lambda, S^i_{j,k} : \tau_i \to s_k, \\ \text{each } v_\iota \text{ fresh from } V_{s_\iota} \end{array}$$

The abstraction or *flattening* rules assign a new abstraction variable to every sub-term in the original set of literals. Abstraction variables are then used as place-holders or equivalence class representatives for those sub-terms. While we would not expect a practical implementation to actually introduce these variables, it greatly simplifies the presentation of the remaining rules. Notice that in each case, a labeling pair for the introduced variables is also created. This corresponds to labeling each sub-term with the set of possible constructors with which it could have been constructed. Also notice that in the **Abstract 3** rule, whenever a selector $S^i_{j,k}$ is applied, we effectively introduce all possible applications of selectors associated with the same constructor. This simplifies the later selector rules and corresponds to the step in the Oppen algorithm which ensures that in the term graph, any node with children has a complete set of children.

*Literal level rules*

**Orient** $$\frac{u \approx v,\ E}{u \to v,\ E} \quad \text{if } \ u \succ v$$

**Remove 1** $$\frac{\mathit{isC}^i_j\, v,\ E}{v \mapsto \{C^i_j\},\ E}$$

**Inconsistent** $$\frac{v \not\approx v,\ E}{\bot}$$

**Remove 2** $$\frac{\neg \mathit{isC}^i_j\, v,\ E}{v \mapsto \mathit{lbls}(\mathit{sort}(v)) \setminus \{C^i_j\},\ E}$$

The simple literal level rules are mostly self-explanatory. The **Orient** rule is used to replace an equation between abstraction variables (which every equation eventually becomes after applying the abstraction rules) with an oriented equation. Oriented equations are used in the remaining rules below. The **Remove** rules remove applications of testers and replace them with labeling pairs that impose the same constraints.

*Upward (i.e., congruence) closure rules*

$$\textbf{Simplify 1} \quad \frac{u \bowtie t,\ u \to v,\ E}{v \bowtie t,\ u \to v,\ E} \qquad \textbf{Simplify 2} \quad \frac{f\boldsymbol{u}u\boldsymbol{v} \to w,\ u \to v,\ E}{f\boldsymbol{u}v\boldsymbol{v} \to w,\ u \to v,\ E}$$

$$\textbf{Superpose} \quad \frac{t \to u,\ t \to v,\ E}{u \to v,\ t \to v,\ E} \quad \text{if } u \succ v$$

$$\textbf{Compose} \quad \frac{t \to v,\ v \to w,\ E}{t \to w,\ v \to w,\ E}$$

These rules are modeled after similar rules for abstract congruence closure in [1]. The **Simplify** and **Compose** rules essentially provide a way to replace any abstraction variable with a smaller (according to $\succ$) one if the two are known to be equal. The **Superpose** rule merges two equivalence classes if they contain the same term. Congruence closure is achieved by these rules because if two terms are congruent, then after repeated applications of the first set of rules, they will become syntactically identical. Then the **Superpose** rule will merge their two equivalence classes.

*Downward (i.e., unification) closure rules*

$$\textbf{Decompose} \quad \frac{C^i_j\boldsymbol{u} \to v,\ C^i_j\boldsymbol{v} \to v,\ E}{C^i_j\boldsymbol{u} \to v,\ \boldsymbol{u} \to \boldsymbol{v},\ E}$$

$$\textbf{Clash} \quad \frac{c \to v,\ d \to v,\ E}{\bot} \quad \text{if } c, d \in \Sigma^{\mathrm{C}},\ c : \sigma,\ d : \sigma,\ c \neq d$$

$$\textbf{Cycle} \quad \frac{C^{i_n}_{j_n}\boldsymbol{u_n}u\boldsymbol{v_n} \to u_{n-1}, \ldots, C^{i_2}_{j_2}\boldsymbol{u_2}u_2\boldsymbol{v_2} \to u_1, C^{i_1}_{j_1}\boldsymbol{u_1}u_1\boldsymbol{v_1} \to u, E}{\bot} \quad \text{if } n \geq 1$$

The main downward closure rule is the **Decompose** rule: whenever two terms with the same constructor are in the same equivalence class, their arguments must be equal. The **Clash** rule detects instances of terms that are in the same equivalence class that must be disequal in the intended model. The **Cycle** rule detects the (inconsistent) cases in which a term would have to be cyclical.

*Selector rules*

$$\textbf{Instantiate 1} \quad \frac{S^i_{j,1}u \to u_1, \ldots, S^i_{j,n^i_j}u \to u_{n^i_j}, u \mapsto \{C^i_j\}, E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u \mapsto \{C^i_j\},\ \mathit{Inst}(u),\ E} \quad \text{if } \mathit{Inst}(u) \notin E$$

$$\textbf{Instantiate 2} \quad \frac{v \mapsto \{C^i_j\},\ E}{\begin{array}{l} C^i_j u_1 \cdots u_{n^i_j} \to v,\ \mathit{Inst}(v),\ E \\ u_1 \mapsto \mathit{lbls}(s^i_{j,1}), \ldots, u_{n^i_j} \mapsto \mathit{lbls}(s^i_{j,n^i_j}) \end{array}} \quad \text{if} \quad \begin{array}{l} \mathit{Inst}(v) \notin E, \\ v \mapsto L \notin E, \\ C^i_j \text{ finite constructor}, \\ S^a_{b,c}(v) \to v' \notin E, \\ u_k \text{ fresh from } V_{s^i_{j,k}} \end{array}$$

$$\textbf{Collapse 1} \quad \frac{C^i_j u_1 \cdots u_{n^i_j} \to u,\ S^i_{j,k}u \to v,\ E}{C^i_j u_1 \cdots u_{n^i_j} \to u,\ u_k \approx v,\ E}$$

$$\textbf{Collapse 2} \quad \frac{S^i_{j,k}u \to v,\ u \mapsto L,\ E}{t^i_{j,k} \approx v,\ u \mapsto L,\ E} \quad \text{if } C^i_j \notin L$$

Rule **Instantiate 1** is used to eliminate selectors by replacing the argument of the selectors with a new term constructed using the appropriate constructor. Notice that only terms that have selectors applied to them can be instantiated and then only once they are unambiguously labeled. Rule **Instantiate 2** is used for finite constructors. For completeness, terms labeled with finite constructors must be instantiated even when no selectors are applied to them. The **Collapse** rules eliminate selectors when the result of their application can be determined. In **Collapse 1**, a selector is applied to a term known to be equal to a constructor of the "right" type. In this case, the selector expression is replaced by the appropriate argument of the constructor. In **Collapse 2**, a selector is applied to a term which must have been constructed with the "wrong" constructor. In this case, the designated term $t^i_{j,k}$ for the selector replaces the selector expression.

*Labeling rules*

$$\textbf{Refine} \quad \frac{v \mapsto L_1,\ v \mapsto L_2,\ E}{v \mapsto L_1 \cap L_2,\ E} \qquad \textbf{Empty} \quad \frac{v \mapsto \emptyset,\ E}{\bot} \quad \text{if } v : \tau_i$$

$$\textbf{Split 1} \quad \frac{S^i_{j,k}(u) \to v,\ u \mapsto \{C^i_j\} \cup L,\ E}{S^i_{j,k}(u) \to v,\ u \mapsto \{C^i_j\},\ E \qquad S^i_{j,k}(u) \to v,\ u \mapsto L,\ E} \quad \text{if } L \neq \emptyset$$

$$\textbf{Split 2} \quad \frac{u \mapsto \{C_j^i\} \cup L,\ E}{u \mapsto \{C_j^i\},\ E \qquad u \mapsto L,\ E} \quad \text{if} \quad \begin{array}{l} L \neq \emptyset, \\ \{C_j^i\} \cup L \text{ all finite constructors} \end{array}$$

The **Refine** rule simply combines labeling constraints that may arise from different sources for the same equivalence class. **Empty** enforces the constraint that every $\tau$-term must be constructed by some constructor. The splitting rules are used to refine the set of possible constructors for a term and are the only rules that cause branching. If a term labeled with only finite constructors cannot be eliminated in some other way, **Split 2** must be applied until it is labeled unambiguously. For other terms, the **Split 1** rule only needs to be applied to distinguish the case of a selector being applied to the "right" constructor from a selector being applied to the "wrong" constructor. On either branch, one of the **Collapse** rules will apply immediately. We discuss this further in Section 4, below. The rules are proved sound, complete and terminating in our full report [4].

# 4 Strategies and Efficiency

It is not difficult to see that the problem of determining the satisfiability of an arbitrary set of literals is NP-complete. The problem was shown to be NP-hard in [12]. To see that it is in NP, we note that given a type completion, no additional splits are necessary, and the remaining rules can be carried out in polynomial time. However, as with other NP-complete problems (Boolean satisfiability being the most obvious example), the right strategy can make a significant difference in practical efficiency.

## *4.1 Strategies*

A *strategy* is a predetermined methodology for applying the rules. Before discussing our recommended strategy, it is instructive to look at the closest related work. Oppen's original algorithm is roughly equivalent to the following: After abstraction, apply the selector rules to eliminate all instances of selector symbols. Next, apply upward and downward closure rules (the bidirectional closure). As you go, check for conflicts using the rules that can derive $\bot$. We will call this the *basic strategy*. Note that it excludes the splitting rules: because Oppen's algorithm assumes a single constructor, the splitting rules are never used. A generalization of Oppen's algorithm is mentioned in [14]. They add the step of initially guessing a "type completion". To model this, consider the following simple **Split** rule:

$$\textbf{Split} \quad \frac{u \mapsto \{C_j^i\} \cup L,\ E}{u \mapsto \{C_j^i\},\ E \qquad u \mapsto L,\ E} \quad \text{if} \quad L \neq \emptyset$$

Now consider a strategy which invokes **Split** greedily (after abstraction) until it no longer applies and then follows the basic strategy. We will call this strategy the *greedy splitting* strategy. One of the key contributions of this paper is to recognize

that the greedy splitting strategy can be improved in two significant ways. First, the simple **Split** rule should be replaced with the smarter **Split 1** and **Split 2** rules. Second, these rules should be delayed as long as possible. We call this the *lazy splitting* strategy. The lazy strategy reduces the size of the resulting derivation in two ways. First, notice that **Split 1** is only enabled when some selector is applied to $u$. By itself, this eliminates many needless case splits. Second, by applying the splitting rules *lazily* (in particular by first applying selector rules), it may be possible to avoid splitting completely in many cases.

**Example.**

Consider the following simple *tree* data type: $\mathit{tree} := \mathit{node}(\mathit{left} : \mathit{tree},\ \mathit{right} : \mathit{tree}) \mid \mathit{leaf}$ with *leaf* as the designated term for both selectors. Suppose we receive the input formula $\mathit{left}^n(Z) \approx X \wedge \mathit{is\_node}(Z) \wedge Z \approx X$. After applying all available rules except for the splitting rules, the result will look something like this:

$$\{\ Z \to u_0, X \to u_0, u_0 \mapsto \{\mathit{node}\}, \mathit{node}(u_1, v_1) \to u_0, u_n \to u_0,$$
$$\mathit{left}(u_1) \to u_2, \ldots, \mathit{left}(u_{n-1}) \to u_n, u_1 \mapsto \{\mathit{leaf}, \mathit{node}\}, \ldots, u_n \mapsto \{\mathit{leaf}, \mathit{node}\},$$
$$\mathit{right}(u_1) \to v_2, \ldots, \mathit{right}(u_{n-1}) \to v_n, v_1 \mapsto \{\mathit{leaf}, \mathit{node}\}, \ldots, v_n \mapsto \{\mathit{leaf}, \mathit{node}\}\},$$

Notice that there are $2n$ abstraction variables labeled with two labels each. If we eagerly applied the naive **Split** rule at this point, the derivation tree would reach size $O(2^{2n})$.

Suppose, on the other hand, that we use the lazy strategy. First notice that **Split 1** can only be applied to $n$ of the abstraction variables $(u_i, 1 \leq i \leq n)$. Thus the more restrictive side-conditions of **Split 1** already reduce the size of the problem to at worst $O(2^n)$ instead of $O(2^{2n})$. However, by only applying it lazily, we do even better: suppose we split on $u_i$. The result is two branches, one with $u_i \mapsto \{\mathit{node}\}$ and the other with $u_i \mapsto \{\mathit{leaf}\}$. The second branch induces a cascade of (at most $n$) applications of Collapse 2 which in turn results in $u_k \mapsto \{\mathit{leaf}\}$ for each $k > i$. This eventually results in $\bot$ via the Empty and Refine rules. The other branch contains $u_i \mapsto \{\mathit{node}\}$ and results in the application of the Instantiate rule, but little else, and so we will have to split again, this time on a different $u_i$. This process will have to be repeated until we have split on all of the $u_i$. At that point, there will be a cycle from $u_0$ back to $u_0$, and so we will derive $\bot$ via the Cycle rule. Because each split only requires at most $O(n)$ rules and there are $n - 1$ splits, the total size of the derivation tree will be $O(n^2)$.

### *4.2 Experimental Results*

We have implemented both the lazy and the greedy splitting strategies in the theorem prover CVC Lite [2]. Using the mutually recursive data types *nat*, *list*, and *tree* mentioned in the introduction, we randomly generated 8000 benchmarks. [5]

[5] See http://www.cs.nyu.edu/~barrett/datatypes for details on the benchmarks and results.

| Worst Case Splits | Num. of Tests | Sat | Unsat | Greedy | | Lazy | |
|---|---|---|---|---|---|---|---|
| | | | | Splits | Time (s) | Splits | Time (s) |
| 0 | 4416 | 306 | 4110 | 0 | 24.6 | 0 | 24.6 |
| 1-5 | 2520 | 2216 | 304 | 6887 | 16.8 | 2414 | 17.0 |
| 6-10 | 692 | 571 | 121 | 4967 | 5.8 | 1597 | 5.7 |
| 11-20 | 178 | 112 | 66 | 2422 | 2.3 | 517 | 1.6 |
| 21-100 | 145 | 73 | 72 | 6326 | 4.5 | 334 | 1.1 |
| 101+ | 49 | 11 | 38 | 16593 | 9.8 | 73 | 0.3 |

Table 1
Greedy vs. Lazy Splitting

As might be expected with a large random set, most of the benchmarks are quite easy. In fact, over half of them are solved without any case splitting at all. However, a few of them did prove to be somewhat challenging (at least in terms of the number of splits required). Table 1 shows the total time and case splits required to solve the benchmarks. The benchmarks are divided into categories based on the the maximum number of case splits required to solve the benchmark.

For easy benchmarks that don't require many splits, the two algorithms perform almost identically. However, as the difficulty increases, the lazy strategy performs much better. For the hardest benchmarks, the lazy strategy outperforms the greedy strategy by more than an order of magnitude.

# 5 Extending the Algorithm

In this section we briefly discuss several ways in which our algorithm can be used as a component in solving a larger or related problem.

## *5.1 Finite Sorts*

Here we consider how to lift the limitation that each of $\sigma \in \{\sigma_1, \ldots, \sigma_r\}$ is infinite valued. Since we have no such restrictions on sorts $\tau_i$, the idea is to simply replace such a $\sigma$ by a new $\tau$-like sort $\tau_\sigma$, whose set of constructors (all of which will be nullary) will match the domain of $\sigma$. For example, if $\sigma$ is a finite scalar of the form $\{1, \ldots, n\}$, then we can let $\tau_\sigma ::== null_1 \mid \ldots \mid null_n$. We then proceed as before, after replacing all occurrences of $\sigma$ by $\tau_\sigma$ and each $i$ by $null_i$.

## *5.2 Simulating Partial Function Semantics*

As mentioned earlier, it is not clear how best to interpret the application of a selector to the wrong constructor. One compelling approach is to interpret selectors as partial functions. An evaluation of a formula then has three possible outcomes:

true, false, or undefined. This approach may be especially valuable in a verification application in which application of selectors is required to be guarded so that no formula should ever be undefined. This can easily be implemented by employing the techniques described in [6]: given a formula to check, a special additional formula called a type-correctness condition is computed (which can be done in time and space linear in the size of the input formula). These two formulas can then be checked using a decision procedure that interprets the partial functions (in this case, the selectors) in some arbitrary way over the undefined part of the domain. The result can then be interpreted to reveal whether the formula would have been true, false, or undefined under the partial function semantics.

### *5.3 Cooperating with other Decision Procedures*

A final point is that that our procedure has been designed to integrate easily into a Nelson-Oppen-style framework for cooperating decision procedures [10]. In the many-sorted case, the key theoretical requirements (see [13]) for two decision procedures to be combined are that the signatures of their theories share at most sort symbols and each theory is *stably infinite* over the shared sorts.[6] A key operational requirement is that the decision procedure is also able to easily compute and communicate equality information.

The theory of $\mathcal{R}$ (i.e., the set of sentences true in $\mathcal{R}$) is trivially stably infinite over the sorts $\sigma_1, \ldots, \sigma_r$ and over any $\tau$-sort containing a non-finite constructor—as all such sorts denote infinite sets in $\mathcal{R}$. Also, in our procedure the equality information is eventually completely captured by the oriented equations produced by the derivation rules, and so entailed equalities can be easily detected and reported.

For a detailed and formal discussion of how to integrate a rule-based decision procedure such as this one into a general framework combining Boolean reasoning and multiple decision procedures, we refer the reader to our related work in [3]. Note that, in particular, this work shows how the internal theory case splits can be delegated on demand to the Boolean engine; this is the implementation strategy followed in CVC Lite.

## References

[1] L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *JAR*, 31:129–168, 2003.

[2] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of CAV*, pages 515–518, July 2004.

[3] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting on demand in satisfiability modulo theories. Technical report, University of Iowa, 2006. Available at ftp://ftp.cs.uiowa.edu/pub/tinelli/papers/BarNOT-RR-06.pdf.

[4] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for satisfiability in the theory of recursive data types. Technical Report TR2005-878, Department of Computer Science, New York University, Nov. 2005.

[6] A many-sorted theory $T$ is stably infinite over a subset $S$ of its sorts if every quantifier-free formula satisfiable in $T$ is satisfiable in a model of $T$ where the sorts of $S$ denote infinite sets.

[5] M. P. Bonacina and M. Echenim. Generic theorem proving for decision procedures. Technical report, Università degli studi di Verona, 2006. Available at http://profs.sci.univr.it/~echenim/.

[6] S. B. et al. A practical approach to partial functions in CVC Lite. In W. A. et al., editor, *Selected Papers from PDPAR '04*, volume 125(3) of *ENTCS*, pages 13–23. Elsevier, July 2005.

[7] W. Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[8] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[9] K. Meinke and J. V. Tucker. Universal algebra. In S. Abramsky, D. V. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1. Claredon Press, 1992.

[10] G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.

[11] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, April 1980.

[12] D. C. Oppen. Reasoning about recursively defined data structures. *JACM*, 27(3):403–411, July 1980.

[13] C. Tinelli and C. Zarba. Combining decision procedures for sorted theories. In J. Alferes and J. Leite, editors, *Proceedings of JELIA '04*, volume 3229 of *LNAI*, pages 641–653. Springer, 2004.

[14] T. Zhang, H. B. Sipma, and Z. Manna. Decision procedures for term algebras with integer constraints. In *Proceedings of IJCAR '04 LNCS 3097*, pages 152–167, 2004.

[15] T. Zhang, H. B. Sipma, and Z. Manna. Term algebras with length function and bounded quantifier alternation. In *Proceedings of TPHOLs*, 2004.